MASTER IN COMPUTER SCIENCE PROFESSIONAL FOCUS IN SOFTWARE ENGINEERING

SAT-Based Concolic Testing in Prolog

Fortz, Sophie

Award date:
2019

Awarding institution:
University of Namur

UNIVERSITÉ
DE NAMUR

FACULTÉ
D'INFORMATIQUE

SAT-Based Concolic Testing in Prolog

Sophie FORTZ

RUE GRANDGAGNAGE, 21 ● B-5000 NAMUR(BELGIUM)

Université de Namur
Faculté d'informatique
Année académique 2018–2019

SAT-Based Concolic Testing in Prolog

Sophie FORTZ

Maître de stage : Pr. German VIDAL

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Pr. Wim VANHOOF

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Abstract

Concolic testing has been studied for years in the field of imperative programming. However, we can only find a very few cases where this technique is applied to other paradigms, like logic programming. This master thesis aims to present a full method to apply both concrete and symbolic execution in parallel on Prolog programs. Our approach is based on a new definition of path coverage, specific to logic programming and called "*choice coverage*". This criteria was defined for the first time by Mesnard et al. (2015). We also introduce a prototype implementation of our algorithm.

Keywords: Concolic execution, symbolic execution, software testing, choice coverage, Prolog, logic programming.

Résumé

Le test concolique est étudié depuis de nombreuses années dans le domaine de la programmation impérative. Cependant, nous ne trouvons dans la littérature que très peu de cas où cette méthode a pu être appliquée à d'autres paradigmes, comme par exemple en programmation logique. Ce mémoire a pour objectif de présenter une méthode complète pour appliquer à la fois une exécution concrète et symbolique en parallèle, sur des programmes Prolog. Notre approche est basée sur une nouvelle définition de la couverture de chemins, appelée "*couverture de choix*" et propre à la programmation logique. Ce critère fut défini pour la première fois par Mesnard et al. (2015). Nous introduirons aussi un prototype d'implémentation de notre algorithme.

Mots-Clefs : Exécution concolique, exécution symbolique, test logiciel, choice coverage, Prolog, programmation logique.

Acknowledgements

Before beginning, I'd like to thank all the people who made this master thesis what it is.

First, I thank Pr. Vanhoof for following me this all year, coming with this subject which perfectly matched my expectations. Thanks to him, I've learned a lot about scientific writing.

Then, I would like to thank Pr. German Vidal, for these incredible three months in Valencia, but also for all his help when I came back in Belgium: re-readings at the speed of light and very wise advice.

I especially thank my lab mates in Valencia, for their welcoming, our coffee breaks and our afternoons games.

Merci à mes frères, tout simplement d'être vous ! Pour votre joie de vivre, même dans les moments plus difficile et quand 1600 (ou 2600) km nous séparent.

Et pour finir, *last but not least*, j'aimerais remercier mes parents, pour leur soutien indéfectible. Si je suis là où j'en suis aujourd'hui, c'est grâce à vous. Pour votre confiance à mon retour, même si c'est parfois difficile de ne rien dire face à mes méthode de travail différentes des vôtres et pour vos encouragement pendant ces cinq, ou plutôt ces vingt-deux dernières années.

Contents

Abstract **v**

Résumé **v**

Acknowledgements **vii**

1 Introduction **1**

2 State of the Art **5**
2.1 What's Concolic Testing? 5
2.2 Concolic Testing for Logic Programming 7
2.2.1 Different paradigms 7
2.2.2 Logic Testing 8
2.2.3 A Recent History of Concolic Testing 8
2.2.4 Prolog Mechanisms 9
2.2.5 Basic Concepts for Concolic Testing 11

3 SAT Solver **13**
3.1 Why Using a Solver? 13
3.2 Microsoft's Z3 Solver 13
3.3 Prolog Interface 14

4 Concolic Algorithm **15**
4.1 Concrete Logic Semantics 15
4.2 Concolic Execution Semantics 18
4.3 Concolic Testing Procedure 25
4.4 Complete Execution 28

5 Ongoing Implementation **39**
5.1 The Working Environment 39
5.2 The Core Implementation 40
5.3 How to Use It? 42

6 Conclusion and Future Developments **43**

Bibliography **44**

Chapter 1

Introduction: From Symbolic to Concolic Execution

Software *validation* is a vast field of computer science. It aims to ensure compliance between implementation and requirements by applying techniques such as testing. A *test case* is a set of input data to be executed by the software. Testing is a popular approach that consists in creating a *test suite* (a list of test cases) and then running the system with these different test cases to observe its behaviour. Figure 1.1 summarize all the activities composing software testing.

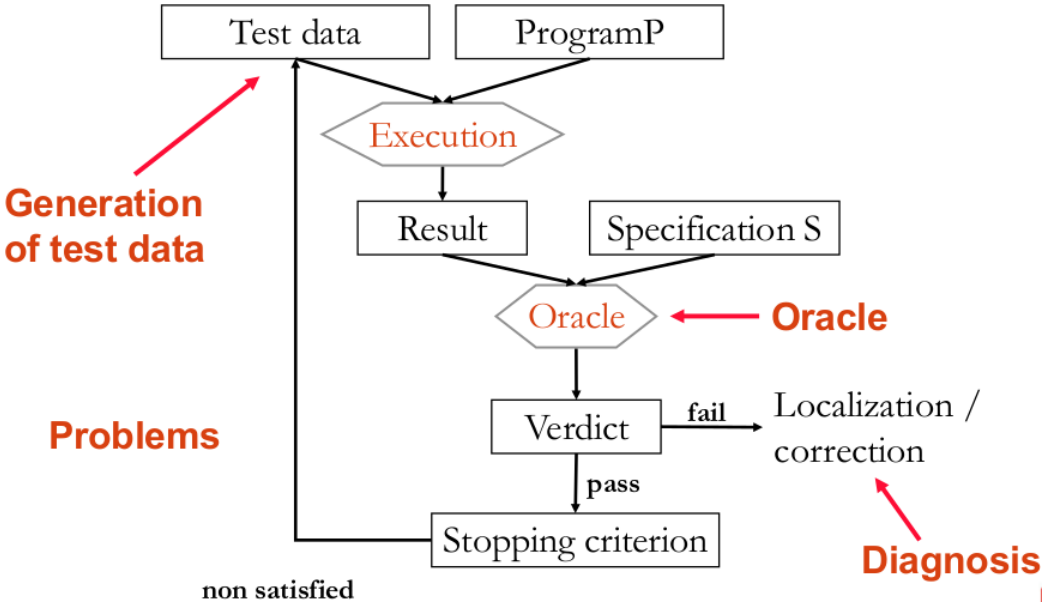

Figure 1.1: Testing activities (Le Traon 2018)

We could discuss each of the activities presented in Figure 1.1, but in this thesis, we will concentrate on the generation of test data. We assume that the *oracle* (the mechanism for determining whether the output is correct) is manual: the user needs to look at the output and tell himself if it is the expected result.

One of the greatest challenges for testing is to define the test cases to execute. This complex and time-consuming task is however necessary because the program's input domain is often very large, or even infinite (like integers for example). It is thus impossible to test all the possible values. So, we need to define a test suite with "good" test cases.

Most of the time, we use the notion of *coverage* (Le Traon 2018) to define the relevance of the test cases. *"Covered"* by a test means being executed at least by one test case in the test suite. To assess the quality of our test, we first need to define a *coverage criteria*. To help us, we can use a *control flow graph*. There exist three major criteria (Figure 1.2):

- Statements or node coverage: Each instruction is performed at least one time.
- Decision or branch coverage: For every choice (intersection in the control graph), all the possible options are taken at least once.
- Path coverage: Cover all the possible sequences of edges in the graph.

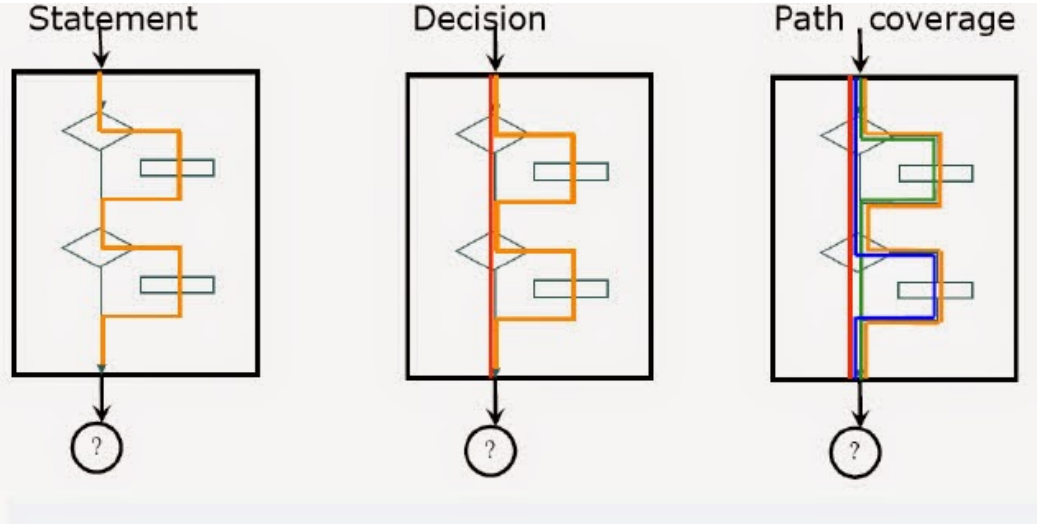

Figure 1.2: Example of different coverage criteria (Learn Testing 2019)

In this thesis, we use the definition of path coverage and adapt it to our needs. Our goal is then to define a test suite which cover as many execution paths as possible.

```
1  int f() {
2    ...
3    y = read();
4    z = y * 2;
5    if (z == 12) {
6      fail();
7    } else {
8      printf("OK");
9    }
10 }
```

Listing 1.1: Code used to illustrate symbolic execution

Among the most popular testing techniques, we can also find symbolic path execution (Wikipedia 2019b; King 1976; Clarke 1976). The idea is to execute the program with symbolic input values. Each statement is executed, creating a new equation. When a conditional statement is reached, the program forks into two new branches. In the example shown in Listing 1.1, we could name the symbolic input λ and then we obtain the two symbolic execution paths from Figure 1.3.

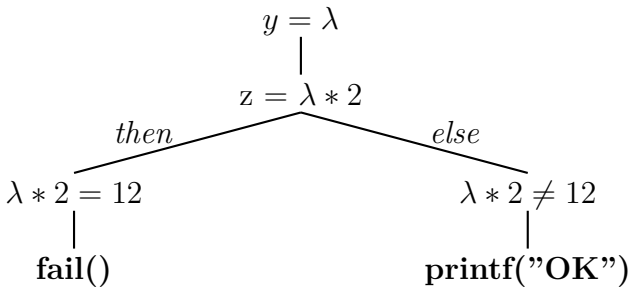

Figure 1.3: Symbolic execution tree

We can now derive a set of constraints associated to each path:

$$\begin{cases} y & = \lambda \\ z & = \lambda * 2 \\ \lambda * 2 & = 12 \end{cases} \qquad \begin{cases} y & = \lambda \\ z & = \lambda * 2 \\ \lambda * 2 & \neq 12 \end{cases}$$

Test cases are found by solving those equation systems. For example, $\lambda = 6$ fails and $\lambda = 5$ prints *"OK"*. These two cases lead to a full path

coverage.

Unfortunately, real-life cases are not so easy. Sets of constraints quickly gain in complexity until becoming unsolvable, leading to poor coverage.

To avoid drawbacks of the symbolic execution, one could think of randomly generating the test cases. Unfortunately, this leads to very poor coverage. Nevertheless, some hybrid approaches where the user drives a random generation exist, such as QuickCheck (Claessen et al. 2000). Although, such methods are again time-consuming.

A quite recent dynamic variant of symbolic execution, called concolic execution (Godefroid, Klarlund, et al. 2005; Sen et al. 2005; Wikipedia 2019a), mixes symbolic and concrete execution. It could be used either for model checking or for test case generation. Using real values helps simplifying the conditions to solve and detects earlier the unfeasible paths.

In this master thesis, we present a new algorithm for concolic testing in Prolog. We will first introduce concolic testing in general, some Prolog specificities and why it is interesting to apply concolic methods on declarative program languages. Then, we will discuss the interest of a SAT solver in our context. The fourth chapter is devoted to development of our algorithm, finishing of a complete step-by-step execution. After describing our proof-of-concept implementation, we will finally conclude by sharing some ideas for future developments.

Chapter 2

State of the Art

After describing how concolic testing works, we discuss its applications in different paradigms and explain why it is interesting to apply it on logic programming.

2.1 What's Concolic Testing?

The main idea of this method is to replace some constraints with a concrete value, reducing their complexity. This simplification helps building more scalable tools. Many tools are already using concolic execution to test imperative programs as for example SAGE (Godefroid, Levin, et al. 2012) and Java Pathfinder (Pasareanu et al. 2010).

```
1 void f(int x, int y) {
2     int z = 2*y;
3     if (x == 100000) {
4         if (x < z) {
5             assert(0); /* error */
6         }
7     }
8 }
```

Listing 2.1: Code used to illustrate concolic execution

To explain how it works, let's take the small C code (Wikipedia 2019a) displayed in Listing 2.1. Our goal is to find a suite of relevant test cases. In this case, line 5 produce an error, since assert(0) is always false. We will show how concolic testing highlights this error.

The execution begins with an arbitrary input instantiation. For example, $x = 0$ and $y = 0$. By executing the first step in concrete and symbolic

execution in parallel, we obtain:

$$\langle z = 0 \,[\![\, z = 2 * y \rangle$$

where the first part of the tuple represent the concrete execution step and the second the symbolic one. Then, the test $x = 100000$ fails ($0 \neq 100000$). This inequality is called a path condition. Every execution following the same path verifies it. At this point, the end of this particular path is reached (no more instruction to execute). We can add the initial values to the list of produced test cases and try new values.

To find these new values, we go back to the last path condition encountered and we negate it. In our case, there is only one: $x \neq 100000$. Now, we need to solve the given constraint $x = 100000$ in order to find values for the input x and y. This step could be made by an automated solver.

Suppose we found $x = 100000$ and $y = 0$ and we run the program again with these new inputs. Since $100000 \geq 0 (= z)$, the program stops and $(x = 100000, y = 0)$ is added to the test cases, before backtracking to the last path condition: $x \geq z$ (with $x = 100000$). The negation of this path condition produces the following set of constraints:

$$\begin{cases} z = 2 * y \\ x = 100000 \\ x < z \end{cases}$$

At this point, we can find the third and last test case of this example: $x = 100000$ and $y = 50001$. This input reaches the erroneous assertion. Since all path conditions have been explored, this is also the end of this concolic execution, summarized in Figure 2.1.

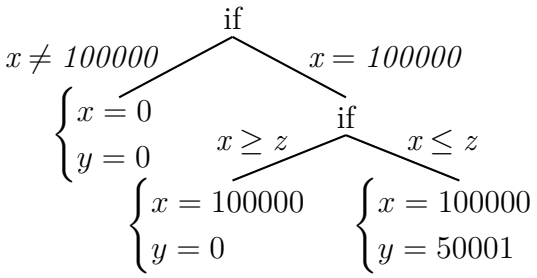

Figure 2.1: Concolic execution tree

The tree shown by Figure 2.1 illustrates the creation of three test cases:

$$\begin{cases} x = 0 \\ y = 0 \end{cases} \quad \begin{cases} x = 100000 \\ y = 0 \end{cases} \quad \begin{cases} x = 100000 \\ y = 50001 \end{cases}$$

Our goal of full path coverage is now satisfied and the third test case covers the error. If only concrete execution had been used (with random values), it could have taken a long time before finding relevant values for all the branches. The error is likely not to be found.

2.2 Concolic Testing for Logic Programming

In this section, we develop research on concolic testing applied to other paradigms. We focus mainly on the logical paradigm with a small detour through functional programming.

2.2.1 Different paradigms

Before going further, here is a small reminder about different programming paradigms. In computer science, There exist two major families of programming languages: imperative and declarative languages.

Imperative programming languages, for example Fortran (Backus et al. 1957), use instructions to modify the state of the program. A state is like a snapshot on the value of every program variable. Imperatives programming describes *how* a program operates. Procedural (Pascal, C, Fortran, etc.) and object-oriented languages (Java, C++, C#, etc.) are two imperative paradigms.

On the other hand, declarative programming aims to explain the logic of the computation and the problem domain, rather than describing how to accomplish it with a sequence of explicit statements. The "how" is left to the implementation of the language. Declarative programming include functional and logic paradigms.

Functional programming (McCarthy 1959) avoids changing-state and mutable data by treating computation as the evaluation of mathematical functions. Functional code avoid all side effects: a function's return value depends only on its arguments. In contrast, in imperative, global program state can affect a function's output in addition to its arguments. The calculation model is closer to the specification of the problem and uses a more abstract level. Functional programs are generally more concise, easier to understand and maintain, quicker to develop and less prone to errors. Haskell (Jones 2003) is a well-known functional language.

We finally arrive at logic programming (Colmerauer et al. 1996; Kowalski 1988; Clocksin et al. 2012), which is the paradigm that concerns us here.

Logic programming is largely based on formal logic. Any program written in a logic programming language is a set of sentences in logical form, ex-

pressing facts and rules about some problem domain. A logic language needs a mechanism of execution which makes it possible to compute these rules. The main advantages of logic programs are that they allow non-determinism (multiple solutions), partial relations and multi-directionality (functions and their inverse). A leader in this paradigm is the Prolog language which is used in this thesis.

2.2.2 Logic Testing

Before talking about *concolic execution* in logic programming, we focus on testing in general for such languages.

PROTest (Belli et al. 1993) is a testing environment for logic programs. It is one of the first tools describing a method directly designed for declarative programming and not adapting imperative ideas.

Mercury (Somogyi et al. 1996) is a logic and functional language inspired by Haskell and mostly Prolog. Degrave et al. (2008) detailed how to automatically generate test inputs for Mercury programs. It adapts symbolic path execution from existing work for imperative languages, to deal with logic features like symbolic data representation, predicate failure and non-determinism.

Mera et al. (2009) proposed a framework for unit testing and verification of the *Ciao* multi-paradigm system (Hermenegildo et al. 2012). Ciao supports logic programming, and Prolog in particular. SWI-Prolog (Wielemaker et al. 2012) offers a unit testing tool. It can also generate test cases interactively and analyse coverage (percentage of used and failing clauses). The SICStus Prolog dialect (Carlsson et al. 2012) and the ECL^{i}PS^e constraint programming system (Schimpf et al. 2012) can compute the number of times a statement in the code is executed when a given goal is run.

The closest approach (both in time and substance) to concolic testing in logic programming is probably Albert et al. (2014). This article depicts a test case generation technique based on traditional symbolic execution and using a form of statement coverage.

2.2.3 A Recent History of Concolic Testing

Until 2015 (Vidal 2015; Mesnard et al. 2015), in the context of functional and logic programming there were only a few testing techniques and none of them used the concolic method, even if its value had been proven in the imperative paradigm (Sen et al. 2005; Godefroid, Levin, et al. 2012; Pasareanu et al. 2010). Since then, things have evolved gradually.

Vidal (2015) developed a concolic execution semantics for logic programs. At this point, it was only ideas, without any formal result or implementation. In this method, concrete execution is driven first, producing a trace which leads the symbolic execution. This step-by-step and conceptually easy manner, however, brought low quality results. It misses one of the main advantages of concolic execution (Mesnard et al. 2015): using the concrete values in symbolic execution. Vidal (2015) also considered a simple statement coverage.

In Mesnard et al. (2015), a fully automatic scheme for concolic testing in logic programming and contest, a proof-of-concept implementation are presented. This algorithm aimed at helping the programmer to systematically find program bugs and generate test cases with a good code coverage. According to Mesnard et al. (2015), contest was the first fully automatic testing tool for Prolog that aims at full path coverage (here called choice coverage, as we explain later in this chapter). This master thesis is an extension of this article, so it will be our major reference. We will often mention the algorithm it presents to test Prolog programs thanks to a concolic method. The three authors published two other articles (Mesnard et al. 2016; Mesnard et al. 2017) detailing some issues of their approach. We discuss these concerns and how our new algorithm tries to answer them in the next section.

Most of researchers use an augmented interpreter of the language to deal with concrete and symbolic values in parallel. Unlike them, Palacios et al. (2015) employed program instrumentation to produce a sequence of events used to reconstruct the symbolic execution associated with the program.

A few other people (Giantsios et al. 2017; Tikovsky 2017), inspired by Mesnard et al. (2015), developed a similar approach for functional programs. *CutEr* (Tikovsky 2017) is a test tool for the functional subset of Core Erlang and *ccti* (Giantsios et al. 2017) is a test tool for Curry, a language derived from Haskell. Cherep Dragoevich (2016) also used *CutEr* as a medium to define a methodology to apply concolic testing, based on pre and post-conditions.

2.2.4 Prolog Mechanisms

Before introducing some basic concepts useful for concolic testing, we quickly remind how Prolog works. These definitions are adapted from Vanhoof (2013).

Prolog programs are made of two basic elements: clauses and atoms. A clause is a rule, defining (part of) a relationship. It is formed by a head, also called an atom, and a body in a conjunctive form. A clause with no body is called a fact. All the clauses form the Prolog database. Each fact is like a

global parameter, always satisfied.

From an initial *request*, Prolog build a search tree called *SLD tree*, for "Selective Linear Definite". Its nodes are queries and the root is the initial query. The construction of the tree is done in depth first. Prolog execution is based on two mechanisms:

- *Unification*: Allows to instantiate the variables of the request and, thus, to calculate answers in case the request is validated. An atom of the query unifies with the head of a clause if there is a substitution of variables that makes the two atoms identical:
  - A *substitution* is a mapping of variables to objects (called *terms*): $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}$
  - Such a substitution is called a *unifier*. There are often multiple unifiers, but we are usually satisfied with *the most general unifier (M.G.U.)*.
  - *Unfolding* is the application of a substitution on an atom. Its result is the atom where all the variables have been replaced simultaneously with the corresponding terms.
- *Backtracking*: Allows to explore several alternatives when looking for a proof. If the leftmost branch of the tree produces a failure, we try the next branch until there is no more left. Backtracking is the mechanism which go up the tree to find a branch that has not been explored yet.

We call a *derivation* a sequence of queries (beginning at the initial one) such that each successive query is constructed from the previous query by unifying the leftmost atom with the head of one of the clauses. When the *leftmost atom* does not unify with any of the (heads of) rules, the derivation fails. When we get the empty query, the derivation succeeds and the final output is the composition of all the unifiers used throughout the derivation. A derivation can also be infinite.

In this thesis, we call *"dis-unification"* the problem consisting in finding substitution for which the request can not unify with given clauses.

The last concept we need is called "*Groundness*". When a variable is not bound to any value, we called it a *free variable*. A grounded variable is exactly the opposite. A grounded term is a term without any free variable. It is an important concept for testing since we are searching for test cases typically in the form of grounded terms.

Here, we only use these basics definitions. In full Prolog, there are several more advanced concepts such as the *negation* and the *cut* that we don't use here.

2.2.5 Basic Concepts for Concolic Testing

Generally, static verification techniques like symbolic execution are complete. On the other hand, dynamic testing, with concrete input, is sound but as we mentioned earlier, this validation technique is incomplete. Soundness can be defined as the absence of false positives: every test case found is a valid test case for the system under consideration. Incompleteness means that we usually don't achieve full coverage. In this thesis, we pursue the work done in Mesnard et al. (2015) to build a sound but generally incomplete approach.

For this purpose, we reuse the notion of *choice coverage* defined in Mesnard et al. (2015). This concept is a variant of path coverage, adapted for logic and nondeterministic programming only. Take the following program for example:

$$P = \begin{cases} p(a). \\ p(b). \end{cases}$$

When using full path coverage, we would be tempted to consider only two test cases:

$$\{p(a), p(b)\}$$

In order to have a full coverage, the notion of *choice coverage* considers that we need a goal matching not only the first and the second clauses, but also no clauses at all and both:

$$\{p(X), p(a), p(b), p(c)\}$$

where $p(X)$ matches both clauses and $p(c)$ none of them.

In the following, we also discuss *unification* and the problems that arise from it. In the context of choice coverage, unification means that an atom A matches the heads of some clauses, say $H_1, \ldots, H_n$, but does not match the heads of some other clauses, say $H'_1, \ldots, H'_m$ (Mesnard et al. 2015).

Mesnard et al. (2015) suggests an algorithm where unification and disunification are key elements to find alternative goals during concolic execution. Unfortunately, the algorithm proposed to solve the problem called *"selective unification"* is incomplete.

Definition 2.2.1 (Selective Unification Problem) *Let A be an atom with $G \subseteq \mathcal{V}ar(A)$ a set of variables. Let $\mathcal{H}^+$ and $\mathcal{H}^-$ be finite sets of atoms such that all atoms are pairwise variable disjoint with A and unify with A. Then, the selective unification problem for A w.r.t. H^+ , H^- and G is defined as follows:*

$$\mathcal{P}(A, \mathcal{H}^+, \mathcal{H}^-, G) = \left\{ \sigma_{\mathcal{V}ar(A)} \;\middle|\; \begin{array}{ll} & \forall\ H \in \mathcal{H}^+ :\ A\sigma \approx H \\ \wedge & \forall\ H \in \mathcal{H}^- :\ \neg(A\sigma \approx H) \\ \wedge & G\sigma \text{ is grounded.} \end{array} \right\}$$

where:

- $\mathcal{V}ar(A)$ *represents the set of all free variables in* A
- $\sigma_{\mathcal{V}ar(A)}$ *is the substitution* σ *restricted to variables of the atom* A
- $A\sigma$ *is the application of the substitution* σ *on the atom* A
- $\approx$ *denotes the unification*

Mesnard et al. (2016) proposes some refinements to complete the algorithm but with some restriction on the linearity of the atoms. Linear means that it can not contain multiple occurrences of the same variable. In some cases, this restriction can have a significant impact. In Mesnard et al. (2017), they even prove that in most practical cases, this selective unification problem is undecidable for constraint logic programs.

Another problem in the Mesnard et al. (2015) implementation was the repetitions. Lots of computations were repeated several times. By modifying the semantics, we could combine some steps to be more efficient and travel each execution trace only once.

In the next chapters, we introduce a new algorithm to meet those requirements and an implementation under development. This approach could be useful for other programming languages as well, with the help of several transformational approaches like the ones presented in Gómez-Zamalloa et al. (2010).

Chapter 3

SAT Solver

Most software implementing concolic testing are using a SAT solver. We will explain why, then we will describe the one we chose and the changes we made to use it in our context.

3.1 Why Using a Solver?

A SAT problem or boolean satisfiability problem is the problem of determining, if it exists, an interpretation that satisfies a given Boolean formula. A SAT solver is a program which aims to solve this kind of problems.

In Mesnard et al. (2015), only unification and dis-unification are used to produce new alternative goals. This method has been proven not efficient (see the *Selective Unification Problem* in the previous chapter). However, several imperative approaches used SAT solvers to deal with the symbolic part of the execution. In this thesis, we also choose this way and produce set of constraints to solve. When performing symbolic execution, each path creates a new equation. Each of them can be solved efficiently by a SAT solver. The drastic improvement of these solvers, in term of efficiency and expressiveness power (Wikipedia 2019a), is one of the reasons why concolic testing developed so much since 2005.

3.2 Microsoft's Z3 Solver

We have chosen to use the *SAT solver Z3 by Microsoft* (Microsoft Research 2019; Github 2019b). This solver designed by Microsoft as an efficient theorem prover, supporting arithmetic, fixed-size bit-vectors, extensional arrays, data-types, uninterpreted functions and quantifiers. They provide an online tutorial to understand its mechanism, and a few APIs to use Z3 with several

programming languages such as C, C++ and Python. This solver has already been used by others in the context of concolic execution, like for example in SAGE (Godefroid, Levin, et al. 2012).

3.3 Prolog Interface

In order to use Z3 in combination with Prolog, we need a new interface. Using the C API (Github 2019a) as a bridge between Prolog and Z3, we define most of the Z3 functionalities in Prolog. This new interface allows to define Z3-contexts, push and pop scopes and types for integer and Prolog terms. With these tools, it is possible to assert a (string) formula, to check it and to recover the correct model (if any).

Chapter 4

Concolic Algorithm

In this chapter, we focus on the new algorithm. Firstly, we describe the semantics used to define the execution of a logic program and the specificities relative to concolic execution. Then we present the whole test procedure and finally, we show an application of the algorithm with a complete example.

4.1 Concrete Logic Semantics

For the purpose of the algorithm, we first need to define the semantics we use for a concrete execution. In fact, we reuse the one described in Mesnard et al. (2015). This top-down semantics is based upon Ströder et al. (2011) and specific to definite logic programs. One of its particularity is that each state contains all the information it needs to perform the next step, in contrast with SLD approaches like Lloyd (1987). Another one is that we only consider the first solution for the initial goal, following the way Prolog is mostly used.

Before presenting the concrete semantics, let us introduce some auxiliary notions and notations from Mesnard et al. (2015):

- The concrete execution of a program $\mathcal{P}$ is expressed by a transition system whose configurations are $\langle \mathcal{B}^1_{\delta_1} | \ldots | \mathcal{B}^n_{\delta_n} \rangle$, where each $\mathcal{B}$ represents a state.

- A goal $\mathcal{B}^{H \leftarrow \mathcal{B}}_{\delta}$ can be both labelled by:

  - A clause $H \leftarrow \mathcal{B}$ of the program, which is the leftmost clause matching $\mathcal{B}$ (i.e. defining the next unification to perform). For simplifying notation, we sometimes use the corresponding label instead of the clause itself.

  - A substitution δ, which represents the solution so far when it is restricted to the variables of the initial goal only. Since it is a top-down semantics, the general substitution is computed step by step by composing different atomic substitutions.

- There exist two special states, called SUCCESS and FAIL and denoting the end of the current path execution. SUCCESS is used when we have found a substitution such that when the program is called with the initial goal, it eventually stops gracefully (returns true). FAIL represents the case where the program returns false.

- As they are static, the program clauses are considered as global parameters.

- A sequence of states is denoted $S, S', \ldots$

- ϵ represents the empty sequence.

- An initial configuration has the form $\langle A_{id} \rangle$ where A is an atomic goal and *id* is the identity substitution.

We also need the definition of a new function (Mesnard et al. 2015):

Definition 4.1.1 (clauses) *Let A be an atom and $\mathcal{P}$ a logic program. The function* clauses$(A, \mathcal{P})$ *returns the sequence $c_1, \ldots, c_n$ of clauses from $\mathcal{P}$ whose head unifies with A.*

The transition rules (Mesnard et al. 2015) shown in Figure 4.1 proceed as follows:

- Rule success leads to a final state, in case of a successful execution path. SUCCESS is a new constant, indexed with the answer substitution computed in the considered derivation.

- Rule failure produces another final state for failing derivations. It introduces the new constant FAIL. The δ substitution could be used for debugging purposes.

- Rule backtrack is used when the first goal fails but there is at least one alternative goal available.

- Rule choice matches the head of the left most atom (in the SLD tree) with a clause. If any, it produces as many copies of the goal as there are solutions of the clauses function.

$$\text{(success)}\ \frac{}{\langle \mathsf{true}_\delta \,|\, S\rangle \to \langle \textsc{success}_\delta\rangle}$$

$$\text{(failure)}\ \frac{}{\langle (\mathsf{fail}, \mathcal{B})_\delta\rangle \to \langle \textsc{fail}_\delta\rangle}$$

$$\text{(backtrack)}\ \frac{S \neq \epsilon}{\langle (\mathsf{fail}, \mathcal{B})_\delta \,|\, S\rangle \to \langle S\rangle}$$

$$\text{(choice)}\ \frac{\mathsf{clauses}(A, \mathcal{P}) = (c_1, \ldots, c_n) \wedge n > 0}{\langle (A, \mathcal{B})_\delta \,|\, S\rangle \to \langle (A, \mathcal{B})_\delta^{c_1} \,|\, \ldots \,|\, (A, \mathcal{B})_\delta^{c_n} \,|\, S\rangle}$$

$$\text{(choice\_fail)}\ \frac{\mathsf{clauses}(A, \mathcal{P}) = \{\}}{\langle (A, \mathcal{B})_\delta \,|\, S\rangle \to \langle (\mathsf{fail}, \mathcal{B})_\delta \,|\, S\rangle}$$

$$\text{(unfold)}\ \frac{\mathsf{mgu}(A, H_1) = \sigma}{\langle (A, \mathcal{B})_\delta^{H_1 \leftarrow \mathcal{B}_1} \,|\, S\rangle \to \langle (\mathcal{B}_1\sigma, \mathcal{B}\sigma)_{\delta\sigma} \,|\, S\rangle}$$

Figure 4.1: Concrete semantics

- Rule choice_fail returns fail when there is no matching clause. This leads to backtrack if there is another goal, or to failure if not.
- Rule unfold executes the unfolding in case we have a matching clause.

To understand the underlying mechanism with a concrete example, consider the following logic program (Mesnard et al. 2015):

$$\begin{array}{lll} p(s(a)). & q(a). & r(a). \\ p(s(X)) \leftarrow q(X). & q(b). & r(c). \\ p(f(X)) \leftarrow r(X). & & \end{array}$$

Given the initial goal $p(f(X))$, we have the following successful computation (for clarity, we label each step with the applied rule):

$$\begin{array}{lllll} \langle p(f(X))_{id}\rangle & \to^{\mathsf{choice}} & \left\langle p(f(X))_{id}^{p(f(Y))\leftarrow r(Y)}\right\rangle & \to^{\mathsf{unfold}} & \langle r(X)_{id}\rangle \\ & \to^{\mathsf{choice}} & \left\langle r(X)_{id}^{r(a)} \,|\, r(X)_{id}^{r(c)}\right\rangle & \to^{\mathsf{unfold}} & \left\langle \mathsf{true}_{\{X/a\}} \,|\, r(X)_{id}^{r(c)}\right\rangle \\ & \to^{\mathsf{success}} & \left\langle \textsc{success}_{\{X/a\}}\right\rangle & & \end{array}$$

$\{X/a\}$ is the first computed answer for the (successful) derivation of $p(f(X))$. The correctness of the concrete semantics is an easy consequence

of the correctness of the semantics in Ströder et al. (2011). The rules of Mesnard et al. (2015) can be seen as an instance for pure Prolog without negation, where we only compute the first answer for the initial goal.

4.2 Concolic Execution Semantics

Using the concrete semantics defined previously, we now introduce a concolic execution semantics for logic programs. This semantic is inspired by Mesnard et al. (2015). Here is a few unchanged preliminary definitions and notation:

- *Concolic states* have the form $\langle S \rrbracket S' \rangle$, where S and S' are sequences of (possibly labelled) concrete and symbolic goals, respectively. In logic programming, the notion of *symbolic* execution is very natural: the structure of both S and S' is the same, and the only difference is that some atoms might be less instantiated in S' than in S.
- Let Z be a set of atoms, $\mathcal{V}ar(Z)$ represents the set of all free variables in Z. For a substitution θ, $\mathcal{V}ar(\theta) = Dom(\theta) \cup Ran(\theta)$.
- $\overline{o_n}$ denotes the sequence of syntactic objects $o_1, \ldots, o_n$.
- Given an atom A, we let $root(A) = p/n$ if $A = p(\overline{t_n})$.
- We assume that every c clause has a corresponding unique label, which we denote by $\ell(c)$.

To these definitions, we add some new ones:

- Since we use a top-down semantics, the execution path of an initial goal is computed step by step. This path is represented by the labels of each clauses which unify during the execution. A trace π is a list of the labels forming (part of) the execution path of the initial goal. If several different paths lead to the same goal, we only consider the left most one in the SLD tree. For example, considering the goal $p(a)$ and the following program:

```
1  p(a).
2  p(X).
```

  Then, $\pi = [1]$ even if $p(a)$ matches with both clauses.
- The symbol $\bot$ is used in a trace to mark a failure at the end of a path.
- We denote by $\hat{\sigma}$ the constraint formed from the substitution σ. For example, if $\sigma = \{X \to a\}$ then $\hat{\sigma} = (X = a)$.

- Given an atom A with $root(A) = p/n$, an *initial concolic state* has the form:
$$\left\langle A_{id} \,[\![\, p(\overline{X_n})_{id,[],p(\overline{X_n}),\mathsf{true},G} \right\rangle$$
where:
  - $\overline{X_n}$ are different fresh variables
  - id is the identity substitution
  - [ ] is the trace computed so far and since it is the initial state, it is currently empty
  - true is the neutral logic constraint which represents here the fact that we haven't found any constraint to solve yet
  - G is a subset of $\overline{X_n}$ containing the variables we want to ground

We also introduce two auxiliary functions:

Definition 4.2.1 (*neg_constr*) *Let A be an atom, G the set of variables to ground and $\overline{c_n}$ a sequence of clauses. The function $neg\_constr(A, G, \overline{c_n})$ returns the set of negative constraints. A negative constraint is a (quantified) constraint in the form:*

$$\forall\ \mathcal{V}ar(A) \setminus G, \quad \forall\ \mathcal{V}ar(c_k), \quad A \neq \mathsf{head}(c_k)$$

where $c_k \in \overline{c_n}$ and the function head *returns the atom at the head of a clause.*

Here is a small example using this definition.

Example 4.2.1 *Lets take the following Prolog program:*

```
1  p ( s ( a ) , b ) .
2  p ( s (W) , a ) .
3  p ( f (W) , s ( Z ) ) .
```

And let A be the goal $p(X, Y)$, with the set of variables to ground $G = \{X\}$. We aim at producing the constraints telling "A must be different from the first and second clauses". Then, we can call the function:

$$neg\_constr(A, G, [p(s(a), b),\ p(s(X), a)])$$

which returns the following constraints:

$$\begin{aligned} \forall\ Y, \quad \forall\ W, \quad & p(X, Y) \neq p(s(a), b) \\ \wedge \quad & p(X, Y) \neq p(s(W), a) \end{aligned}$$

Definition 4.2.2 (alts) *Let A_0 be the initial concrete goal and A a symbolic one. Let γ be the constraints so far and G the set of variables to ground. Let $\overline{c_k}$ and $\overline{d_k}$ be the sets of clauses matching the concrete and the symbolic goal respectively. Then, the function* $\mathsf{alts}(A_0, \gamma, A, \overline{c_k}, \overline{d_k}, G)$ *returns all the alternative goals to consider. The set of alternatives goals can be defined as:*

$$\mathsf{alts} = \left\{ Goals \;\middle|\; \begin{array}{ll} & \mathcal{P} \in Power\_Set(\overline{d_k}) \\ \wedge & \mathcal{P} \neq \overline{c_k} \\ \wedge & Goals \vdash (\gamma \wedge \hat{\mathcal{P}}). \end{array} \right\}$$

where $\vdash$ can be understand as "satisfies", $Power\_Set$ is the set of all subsets, and $\hat{\mathcal{P}}$ is the positive constraint formed from $\mathcal{P}$, i.e. for an initial goal A:

$$\forall\ clause \in \mathcal{P}, \quad \forall\ \mathcal{V}ar(A) \setminus G, \quad \forall\ \mathcal{V}ar(clause), \quad A = \mathsf{head}(clause)$$
$$\wedge \quad \forall\ clause \in \overline{d_k} \setminus \mathcal{P}, \quad \forall\ \mathcal{V}ar(A) \setminus G, \quad \forall\ \mathcal{V}ar(clause), \quad A \neq \mathsf{head}(clause)$$

where the function head *returns the atom at the head of a clause. In practice, we use the Z3 SAT solver to solve the constraints computing the alternative goals.*

Example 4.2.2 *Lets take the following Prolog program:*

```
1   p ( s ( a ) , b ) .
2   p ( s (X) , a )  :-  q (X) .
3   p ( f (X) , s (Y) ) .

5   q ( a ) .
6   q ( b ) .
```

With $A_0 = p(X, Y)$ the initial goal, a symbolic goal $A = q(W)$, and the set of variables to ground $G = \{X, W\}$. Let the γ constraints be:

$$\begin{array}{rll} \gamma = \forall\ Y, V, Z & p(X, Y) \neq p(s(a), b) \\ & \wedge & p(X, Y) \neq p(f(V), s(Z)) \\ & \wedge & (X = s(W)) \wedge (Y = a) \end{array}$$

We want to find the alternative goals for $\overline{c_k} = [q(b)]$ and $\overline{d_k} = [q(a), q(b)]$. Since we have:

$$Power\_Set(\overline{d_k}) = \{\{\}, \{q(a)\}, \{q(b)\}, \{q(a), q(b)\}\}$$

For each set, we can then make the following deductions:

$$\begin{array}{rlll} \{\} \longrightarrow & \gamma \wedge (q(W) \neq q(a)) \wedge (q(W) \neq q(b)) & \longrightarrow & p(s(c), a) \\ \{q(a)\} \longrightarrow & \gamma \wedge (q(W) = q(a)) \wedge (q(W) \neq q(b)) & \longrightarrow & \textit{no new goal} \\ \{q(b)\} \longrightarrow & q(b) = \overline{c_k} & \longrightarrow & \textit{no new goal} \\ \{q(a), q(b)\} \longrightarrow & \gamma \wedge (q(W) = q(a)) \wedge (q(W) = q(b)) & \longrightarrow & \textit{no new goal} \end{array}$$

where c is a new fresh constant. The constraints do not need any new quantifiers since we don't use another variable than W, which is in G. The output of the function $\mathsf{alts}(A_0, \gamma, A, \overline{c_k}, \overline{d_k}, G)$ *is thus the set:*

$$\{p(s(c), a)\}$$

Finally, we define two global parameters: *Traces* and *TestCases*. The set *Traces* contains all the traces already visited. It needs to be initialized empty. The produced test cases are stored in the *TestCases* set. This global parameter is initialized with the initial concrete goal.

The transition relation $\rightsquigarrow$ has undergone some changes from Mesnard et al. (2015). The new relation is depicted in Figure 4.2.

$$\text{(success)}\ \frac{}{\langle \mathsf{true}_\delta \,|\, S \,[\![\, \mathsf{true}_{\theta,\pi,A_0,\gamma,G} \,|\, S' \rangle \rightsquigarrow \langle \textsc{success}_\delta \,[\![\, \textsc{success}_\theta \rangle}$$

$$\text{(failure)}\ \frac{}{\langle (\mathsf{fail},\mathcal{B})_\delta \,[\![\, (\mathsf{fail},\mathcal{B}')_{\theta,\pi,A_0,\gamma,G} \rangle \rightsquigarrow \langle \textsc{fail}_\delta \,[\![\, \textsc{fail}_\theta \rangle}$$

$$\text{(backtrack)}\ \frac{S \neq \epsilon}{\langle (\mathsf{fail},\mathcal{B})_\delta \,|\, S \,[\![\, (\mathsf{fail},\mathcal{B}')_{\theta,\pi,A_0,\gamma,G} \,|\, S' \rangle \rightsquigarrow \langle S \,[\![\, S' \rangle}$$

$$\text{(choice)}\ \frac{\begin{array}{l}\mathsf{clauses}(A,\mathcal{P}) = \overline{c_n} \wedge n > 0 \wedge \mathsf{clauses}(A',\mathcal{P}) = \overline{d_k} \\ \gamma' \leftarrow \mathit{neg\_constr}(A', \overline{d_k} \setminus \overline{c_n}) \\ \mathit{if}\ \pi \notin \mathit{Traces}\ \mathit{then}\ \mathit{TestCases} \leftarrow \mathit{TestCases} \cup \mathsf{alts}(A_0,\gamma,A',\overline{c_n},\overline{d_k},G) \\ \qquad \mathit{Traces} \leftarrow \mathit{Traces} \cup \pi,\end{array}}{\begin{array}{l}\langle (A,\mathcal{B})_\delta \,|\, S \,[\![\, (A',\mathcal{B}')_{\theta,\pi,A_0,\gamma,G} \,|\, S' \rangle \\ \rightsquigarrow \langle (A,\mathcal{B})^{c_1}_\delta \,|\, \ldots \,|\, (A,\mathcal{B})^{c_n}_\delta \,|\, S \\ \qquad [\![\, (A',\mathcal{B}')^{c_1}_{\theta,\pi.\ell(c_1),A_0,\gamma\wedge\gamma',G} \,|\, \ldots \,|\, (A',\mathcal{B}')^{c_n}_{\theta,\pi.\ell(c_n),A_0,\gamma\wedge\gamma',G} \,|\, S' \rangle\end{array}}$$

$$\text{(choice\_fail)}\ \frac{\begin{array}{l}\mathsf{clauses}(A,\mathcal{P}) = \{\} \wedge \mathsf{clauses}(A',\mathcal{P}) = \overline{c_k} \\ \gamma' \leftarrow \mathit{neg\_constr}(A', \overline{c_k}) \\ \mathit{if}\ \pi \notin \mathit{Traces}\ \mathit{then}\ \mathit{TestCases} \leftarrow \mathit{TestCases} \cup \mathsf{alts}(A_0,\gamma,A',\{\},\overline{c_k},G) \\ \qquad \mathit{Traces} \leftarrow \mathit{Traces} \cup \pi,\end{array}}{\langle (A,\mathcal{B})_\delta \,|\, S \,[\![\, (A',\mathcal{B}')_{\theta,\pi,A_0,\gamma,G} \,|\, S' \rangle \rightsquigarrow \langle (\mathsf{fail},\mathcal{B})_\delta \,|\, S \,[\![\, (\mathsf{fail},\mathcal{B}')_{\theta,\pi.\bot,A_0,\gamma\wedge\gamma',G} \,|\, S' \rangle}$$

$$\text{(unfold)}\ \frac{\mathsf{mgu}(A,H_1) = \sigma \wedge \mathsf{mgu}(A',H_1) = \rho}{\begin{array}{l}\left\langle (A,\mathcal{B})^{H_1 \leftarrow \mathcal{B}_1}_\delta \,|\, S \,[\![\, (A',\mathcal{B}')^{H_1 \leftarrow \mathcal{B}_1}_{\theta,\pi,A_0,\gamma,G} \,|\, S' \right\rangle \\ \rightsquigarrow \left\langle (\mathcal{B}_1\sigma, \mathcal{B}\sigma)_{\delta\sigma} \,|\, S \,[\![\, (\mathcal{B}_1\rho, \mathcal{B}'\rho)_{\theta\rho,\pi,A_0,\gamma\wedge\widehat{\rho},G\wedge\mathcal{V}ar(G\rho)} \,|\, S' \right\rangle\end{array}}$$

Figure 4.2: Concolic execution semantics

As can be seen, the concrete part of the configurations behaves the same way as the concrete semantics presented in the previous section. Here are some explanations about the parallel symbolic execution:

- Rules success and failure behaves like the concrete version: they lead to a final configuration, indexed with the answer substitution computed in the considered (successful or failing) derivation.

- Rule backtrack is also similar for concrete and symbolic states. The idea is just to skip the first failing goal and to continue the derivation

with the second one. The $S \neq \epsilon$ hypothesis ensures the existence of this second goal.

- Rule choice finds matching clauses to pursue the concrete execution and uses the symbolic goals to find new alternatives. The clauses function is used to compute the clauses matching both the concrete and the symbolic goals ($\overline{c_n}$ and $\overline{d_k}$ respectively). Obviously, $\overline{c_n} \subseteq \overline{d_k}$. We also compute the γ' negative constraints for all the clauses matching the symbolic goal but not the concrete one. If the π trace has not already been considered, then it is added to the set *Traces* and alternative goals are computed and added to *TestCases*. The rule creates as many new symbolic goals as it creates concrete goals. The indexes of the symbolic goals are updated as following:
  - The label of the matching clause is added to the π trace of each new symbolic goal.
  - γ' is added to the previous γ constraints.
  - The other indexes stay unchanged.
- Rule choice_fail follows the same idea than rule choice, but in the case where there is no matching clause ($\mathsf{clauses}(A, \mathcal{P}) = \{\}$). So, every clauses matching the symbolic goal must be negated and instead of a clause label, we add the special symbol $\bot$ to the trace.
- Rule unfold still executes the unfolding in case we have a matching clause. The only difference with the concrete execution is that now we need to update the indexes:
  - A $\hat{\rho}$ constraint is added to the previous γ constraints. $\hat{\rho}$ is a constraint formed from the new ρ substitution, where each substituted variable is equalized to the term which unified with it.
  - The new variables used in the ρ substitution are added to set of variables to be grounded if they unify with a variable in G.
  - The other indexes stay unchanged.

Let us take a small example from Mesnard et al. (2015) to apply our new semantics:

Example 4.2.3 *Consider the following logic program with clause labels:*

$$
\begin{array}{lll}
(\ell_1)\ p(s(a)). & (\ell_4)\ q(a). & (\ell_6)\ r(a). \\
(\ell_2)\ p(s(X)) \leftarrow q(X). & (\ell_5)\ q(b). & (\ell_7)\ r(c). \\
(\ell_3)\ p(f(X)) \leftarrow r(X). & &
\end{array}
$$

Given the initial goal $p(f(a))$, we have the following concolic execution:

$$\Big\langle p(f(a))_{id} \,[\![\, p(W)_{id,[],p(W),\mathsf{true},[W]} \Big\rangle$$

$$\textit{choice} \;\Big|\; \overline{c_n} = \{3\}, \overline{d_k} = \{1,2,3\}$$

$$\Big\langle p(f(a))^{p(f(Y))\leftarrow r(Y)}_{id} \,[\![\, p(W)^{p(f(Y))\leftarrow r(Y)}_{id,[3],p(W),\gamma,[W]} \Big\rangle$$

$$\textit{unfold} \;\Big|$$

$$\Big\langle r(a)_{id} \,[\![\, r(Y)_{\{W\to f(Y)\},[3],p(W),\gamma\wedge\hat{\sigma},[W,Y]} \Big\rangle$$

$$\textit{choice} \;\Big|\; \overline{c_n} = \{6\}, \overline{d_k} = \{6,7\}$$

$$\Big\langle r(a)^{r(a)}_{id} \,[\![\, r(Y)^{r(a)}_{\{W\to f(Y)\},[3,6],p(W),\gamma\wedge\hat{\sigma}\wedge\gamma',[W,Y]} \Big\rangle$$

$$\textit{unfold} \;\Big|$$

$$\Big\langle \mathsf{true}_{id} \,[\![\, \mathsf{true}_{\left\{\substack{W\to f(a)\\ Y\to a}\right\},[3,6],p(W),\gamma\wedge\hat{\sigma}\wedge\gamma'\wedge\hat{\sigma}',[W,Y]} \Big\rangle$$

$$\textit{success} \;\Big|$$

$$\Big\langle \textsc{success}_{id} \,[\![\, \textsc{success}_{\left\{\substack{W\to f(a)\\ Y\to a}\right\}} \Big\rangle$$

With:

$$\begin{aligned}
\gamma &= \forall\ X \quad p(W) \neq p(s(a)) \\
&\qquad \wedge \quad p(W) \neq p(s(X)) \\
\hat{\sigma} &= (W = f(Y)) \\
\gamma' &= r(Y) \neq r(b) \\
\hat{\sigma}' &= (Y = a)
\end{aligned}$$

First, we apply a choice *rule. The concrete goal only matches $p(f(X))$, when the symbolic one matches $p(s(a))$, $p(s(X))$ and $p(f(X))$. We can then execute an* unfold *rule to apply the substitution. The* choice *rule called again, this time with $\{r(a)\}$ and $\{r(a),\ r(c)\}$ for the concrete and symbolic goals respectively. After performing a second* unfold *rule, we can use the* success *rule and the end of the execution path is reached. The concolic execution stops.*

This may seem longer than the previous definition (Mesnard et al. 2015). This comes from the fact that more computations are done in each step. It

avoids creating more steps which perform the same computation again and again. In the old version, the semantics was there only for the concolic execution. Here, nearly all the testing is already done in parallel. For simplicity, this example does not take TestCases and Traces global parameters into account, or the result of the alts *function. These are only useful if we wanted to do a full testing procedure and not only a path execution like now. A complete example is available in the last section of this chapter.*

4.3 Concolic Testing Procedure

In this section, we modify the concolic testing procedure developed in Mesnard et al. (2015) to fit the new concolic execution semantics. The presented algorithm aims to produce concrete initial goals so that all *feasible* choices in the execution paths are covered. An implementation of this procedure is described in the next chapter.

Our *concolic testing procedure* (Algorithm 1) takes as input a program, a set G of variables to be grounded and a random initial atomic goal (generally provided by the user) denoted $main/n$. We also define a function input, which associates some *input* arguments to $main/n$.

We assume that each initial concrete goal $main(\overline{t_n})$ execution terminates in a finite number of steps or finitely fails (Vasak et al. 1986).

Algorithm 1: Concolic testing

Input: a logic (labelled) program P, a set G of variables to ground and an atom $main(\overline{t_n})$ with $\mathsf{input}(main(\overline{t_n}))$ ground.

Output: a set *TestCases* of test cases.

1 Let $Pending := \{main(\overline{t_n})\}$, $TestCases := \{\}$, $Traces := \{\}$.

2 **while** $|Pending| \neq 0$ **do**

2.a Take $A \in Pending$, $Pending := Pending \setminus \{A\}$, $TestCases := TestCases \cup \{A\}$.

2.b Apply *Concolic execution rules* on A, with the initial concolic state defined as:

$$\mathcal{C}_0 = \left\langle A_{id} \,[\![\, main(\overline{X_n})_{id,[],p(\overline{X_n}),\mathsf{true},G} \right\rangle$$

2.c Update *Pending* by adding each new test case produced during this execution.

3 **end**

4 Return *TestCases*.

Algorithm 1 begins by initializing the global parameters (*Pending*, *TestCases* and *Traces*), then it considers each pending test case until there are no more left. For every pending test case, a concolic execution is performed, as describe in the previous section. At the end of this execution, we add all the new discovered goals to the set of the pending test cases. When all test cases have been considered, the procedure returns *TestCases*.

The soundness of concolic testing comes from the fact that each atom from *TestCases* is indeed a test case of the form $main(\overline{s_n})$ with $\mathsf{input}(main(\overline{s_n}))$ ground.

Unfortunately, we often need to choose between *completeness* and *termination*. If our algorithm terminates, it produces test cases to cover all feasible paths. In this sense, it could be considered as a complete semi-algorithm (Mesnard et al. 2015). However, to get full choice coverage, the algorithm runs forever if the program involves recursion.

Example 4.3.1 *Let $nat(s(0))$ be an initial goal and consider the following program:*

$$(\ell_1)\ nat(0). \qquad (\ell_2)\ nat(s(X)) \leftarrow nat(X).$$

Here is the concolic trace produced by the initial goal:

$$\begin{array}{c}
\left\langle nat(s(0))_{id} \,[\![\, nat(X)_{id,[],p(X),\mathsf{true},[X]} \right\rangle \\
\textit{choice} \;\Big|\quad \overline{c_n} = \{2\}, \overline{d_k} = \{1,2\} \\
\left\langle nat(s(0))^{(2)}_{id} \,[\![\, nat(X)^{(2)}_{id,[2],p(X),\gamma,[X]} \right\rangle \\
\textit{unfold} \;\Big| \\
\left\langle nat(0)_{id} \,[\![\, nat(X')_{\sigma,[2],p(X),\gamma\wedge\hat{\sigma},[X,X']} \right\rangle \\
\textit{choice} \;\Big|\quad \overline{c_n} = \{1\}, \overline{d_k} = \{1,2\} \\
\left\langle nat(0)^{(1)}_{id} \,[\![\, nat(X')^{(1)}_{\sigma,[2,1],p(X),\gamma\wedge\hat{\sigma}\wedge\gamma',[X,X']} \right\rangle \\
\textit{unfold} \;\Big| \\
\left\langle \mathsf{true}_{id} \,[\![\, \mathsf{true}_{\sigma\sigma',[2,1],p(X),\gamma\wedge\hat{\sigma}\wedge\gamma'\wedge\hat{\sigma}',[X,X']} \right\rangle \\
\textit{success} \;\Big| \\
\left\langle \textsc{success}_{id} \,[\![\, \textsc{success}_{\sigma\sigma'} \right\rangle
\end{array}$$

After applying a first choice *rule, we can produce alternative goals:*

$$\begin{array}{rl}
\{\} & \longrightarrow nat(1) \\
\{1\} & \longrightarrow nat(0) \\
\{2\} & \longrightarrow nat(s(0)) \quad \textit{(already considered)} \\
\{1,2\} & \longrightarrow \textit{no solution}
\end{array}$$

Since the trace $\pi = [2]$ has not been considered yet, the second choice *will also try to generate new goals. We obtain:*

$$\begin{array}{rl}
\{\} & \longrightarrow nat(s(1)) \\
\{1\} & \longrightarrow nat(s(0)) \quad \textit{(already considered)} \\
\{2\} & \longrightarrow nat(s(s(0))) \\
\{1,2\} & \longrightarrow \textit{no solution}
\end{array}$$

With only one concolic execution, we have collected four new goals, in addition to the initial one:

$$\{nat(s(0)), nat(1), nat(0), nat(s(1)), nat(s(s(0)))\}.$$

We will not detail here the concolic execution of each new goal, but we still can make some observations thanks to the structure of the new test cases. If we compare the test cases produced by the first and the second rule, we can see they are related. The only difference is that we have applied s once more time on each test generated by the second choice.

If we had applied the method on the last test case $nat(s(s(0)))$, we would have needed to use three times the choice rule. The two first calls would have produced the same traces (and so, no new goals), but the third call would have generated some new ones like $nat(s(s(s(0))))$ and so on.

The algorithm, if we let it run forever, produces an infinity of test cases, with a trace always longer $([1], [2, 1], [2, 2, 1], [2, 2, 2, 1], etc)$. This is due to the recursivity of the predicate.

In practice, we generally prefer termination over completeness. There exist several techniques to stop the algorithm at some point (Mesnard et al. 2015): a time limit, a bound for the length of concolic executions or a maximum term depth for the arguments of the generated test cases. The implementation described in the next chapter uses this last possibility. This assumption allows to create only a finite number of test cases and thus ensures the termination.

4.4 Complete Execution

This section describes the step-by-step execution of our algorithm on a concrete example. The code displayed in Listing 4.1 is used as the input program. Each clause is referenced by its line number.

```
1  p(s(a),b).
2  p(s(X),a) :- q(X).
3  p(f(X),s(Y)) :- r(X,Y).

5  q(a).
6  q(b).

8  r(a,b).
9  r(c,b).
```

Listing 4.1: Prolog input program

To test this program, we need an initial call provided by the user. Here, we use the following one:

$$\langle p(a, Y)_{id} \rangle\!\rangle\, p(W_1, W_2)_{id,[],p(W_1,W_2),\mathsf{true},[W_1]} \rangle$$

This means that $p(a, Y)$ is the atomic concrete goal and that we want only the first variable (W_1) to be grounded at the end of the execution. Since no substitution has been made, we indexed each goal with *id*. The trace is empty, the first symbolic goal is $p(W_1, W_2)$ and we don't need to take any constraint into account (true).

We also need to initialize the set of traces and the set of test cases with the first concrete goal:

$$\begin{aligned} \mathit{Traces} &\leftarrow \{\} \\ \mathit{TestCases} &\leftarrow \{p(a, Y)\} \end{aligned}$$

We are now ready to start the concolic testing procedure. First, we perform a concolic execution on the initial goal, as shown in Figure 4.3.

$$\begin{array}{c} \langle p(a,Y)_{id} \rrbracket\ p(W_1,W_2)_{id,[],p(W_1,W_2),\mathsf{true},[W_1]} \rangle \\ \text{choice\_fail} \ \Big| \ \ \overline{c_n} = \{\}, \overline{d_k} = \{1,2,3\} \\ \langle \mathsf{fail}_{id} \rrbracket\ \mathsf{fail}_{id,[],p(W_1,W_2),\gamma,[W_1]} \rangle \\ \text{failure} \ \Big| \\ \langle \textsc{fail}_{id} \rrbracket\ \textsc{fail}_{id} \rangle \end{array}$$

$$\begin{aligned} \gamma = \forall\ W_2, X, Y \quad & p(W_1, W_2) \neq p(s(a), b) \\ \wedge \quad & p(W_1, W_2) \neq p(s(X), a) \\ \wedge \quad & p(W_1, W_2) \neq p(f(X), s(Y)) \end{aligned}$$

$$\mathit{Traces} \leftarrow \mathit{Traces} \cup \{[\,]\}$$

$$\mathit{TestCases} \leftarrow \mathit{TestCases} \cup \{p(s(b), W_2),\ p(f(a), W_2),\ p(s(a), W_2)\}$$

Figure 4.3: Execution path of $p(a, Y)$

After performing the first complete execution path (Figure 4.3), we have produced three new test cases: $p(s(b), W_2)$, $p(f(a), W_2)$ and $p(s(a), W_2)$. Figures 4.4, 4.5 and 4.6 respectively show their concolic execution. We can notice in Figure 4.6 that the rule choice leads to two different goals. This is a phenomenon we talked about, when the initial goal matches several clauses. In this case, the first produced goal leads to a successful path, so we don't need to backtrack.

$$\Big\langle p(s(b),Y)_{id} \;[\![\; p(W_1,W_2)_{id,[],p(W_1,W_2),\mathsf{true},[W_1]} \Big\rangle$$

$$\text{choice} \;\Big|\; \overline{c_n} = \{2\}, \overline{d_k} = \{1,2,3\}$$

$$\Big\langle p(s(b),Y)^{(2)}_{id} \;[\![\; p(W_1,W_2)^{(2)}_{id,[2],p(W_1,W_2),\gamma,[W_1]} \Big\rangle$$

$$\text{unfold} \;\Big|$$

$$\Big\langle q(b)_{\{Y\to a\}} \;[\![\; q(W')_{\sigma,[2],p(W_1,W_2),\gamma\wedge\hat{\sigma},[W_1,W']} \Big\rangle$$

$$\text{choice} \;\Big|\; \overline{c_n} = \{6\}, \overline{d_k} = \{5,6\}$$

$$\Big\langle q(b)^{(6)}_{\{Y\to a\}} \;[\![\; q(W')^{(6)}_{\sigma,[2,6],p(W_1,W_2),\gamma\wedge\hat{\sigma}\wedge\gamma',[W_1,W']} \Big\rangle$$

$$\text{unfold} \;\Big|$$

$$\Big\langle \mathsf{true}_{\{Y\to a\}} \;[\![\; \mathsf{true}_{\sigma\sigma',[2,6],p(W_1,W_2),\gamma\wedge\hat{\sigma}\wedge\gamma'\wedge\hat{\sigma}',[W_1,W']} \Big\rangle$$

$$\text{success} \;\Big|$$

$$\Big\langle \textsc{success}_{\{Y\to a\}} \;[\![\; \textsc{success}_{\sigma\sigma'} \Big\rangle$$

$$\begin{aligned}
&\gamma = \forall\, W_2, X, Y \quad p(W_1,W_2) \neq p(s(a),b) \\
&\qquad\qquad\quad \wedge \quad p(W_1,W_2) \neq p(f(X),s(Y)) \\
&\sigma = \{W_1 \to s(W'), \\
&\qquad W_2 \to a\} \\
&\hat{\sigma} = (W_1 = s(W')) \wedge (W_2 = a) \\
&\gamma' = q(W') \neq q(a) \\
&\sigma' = \{W' \to b\} \\
&\hat{\sigma}' = (W' = b) \\
&\mathit{Traces} \leftarrow \mathit{Traces} \cup \{[2],[2,6]\} \\
&\mathit{TestCases} \leftarrow \mathit{TestCases} \cup \{p(s(c),W_2)\}
\end{aligned}$$

Figure 4.4: Execution path of $p(s(b), Y)$

$$\Big\langle p(f(a),Y)_{id} \;[\![\; p(W_1,W_2)_{id,[],p(W_1,W_2),\mathsf{true},[W_1]} \Big\rangle$$

$$\text{choice} \;\Big|\; \overline{c_n} = \{3\}, \overline{d_k} = \{1,2,3\}$$

$$\Big\langle p(f(a),Y)^{(3)}_{id} \;[\![\; p(W_1,W_2)^{(3)}_{id,[3],p(W_1,W_2),\gamma,[W_1]} \Big\rangle$$

$$\text{unfold} \;\Big|$$

$$\Big\langle r(a,Y')_{\{Y\to s(Y')\}} \;[\![\; r(W_1',W_2')_{\sigma,[3],p(W_1,W_2),\gamma\wedge\hat{\sigma},[W_1,W_1',W_2']} \Big\rangle$$

$$\text{choice} \;\Big|\; \overline{c_n} = \{8\}, \overline{d_k} = \{8,9\}$$

$$\Big\langle r(a,Y')^{(8)}_{\{Y\to s(Y')\}} \;[\![\; r(W_1',W_2')^{(8)}_{\sigma,[3,8],p(W_1,W_2),\gamma\wedge\hat{\sigma}\wedge\gamma',[W_1,W_1',W_2']} \Big\rangle$$

$$\text{unfold} \;\Big|$$

$$\Big\langle \mathsf{true}_{\left\{\begin{smallmatrix} Y\to s(b) \\ Y'\to b \end{smallmatrix}\right\}} \;[\![\; \mathsf{true}_{\sigma\sigma',[3,8],p(W_1,W_2),\gamma\wedge\hat{\sigma}\wedge\gamma'\wedge\hat{\sigma}',[W_1,W_1',W_2']} \Big\rangle$$

$$\text{success} \;\Big|$$

$$\Big\langle \textsc{success}_{\left\{\begin{smallmatrix} Y\to s(b) \\ Y'\to b \end{smallmatrix}\right\}} \;[\![\; \textsc{success}_{\sigma\sigma'} \Big\rangle$$

$$\begin{aligned}
&\gamma = \forall\ W_2, X \quad p(W_1,W_2) \neq p(s(a),b) \\
&\qquad\qquad \wedge \quad p(W_1,W_2) \neq p(s(X),a) \\
&\sigma = \{W_1 \to f(W_1'), \\
&\qquad W_2 \to s(W_2')\} \\
&\hat{\sigma} = (W_1 = f(W_1')) \wedge (W_2 = s(W_2')) \\
&\gamma' = \forall\ W_2' \quad r(W_1',W_2') \neq r(c,b) \\
&\sigma' = \{W_1' \to a, \\
&\qquad W_2' \to b\} \\
&\hat{\sigma}' = (W_1' = a) \wedge (W_2' = b) \\
&\mathit{Traces} \leftarrow \mathit{Traces} \cup \{[3],[3,8]\} \\
&\mathit{TestCases} \leftarrow \mathit{TestCases} \cup \{p(f(b),W_2),\ p(f(c),W_2)\}
\end{aligned}$$

Figure 4.5: Execution path of $p(f(a),Y)$

$$\langle p(s(a),Y)_{id} \,[\![\, p(W_1,W_2)_{id,[],p(W_1,W_2),\mathsf{true},[W_1]} \rangle$$

$$\text{choice} \quad \Big| \quad \overline{c_n} = \{1,2\}, \overline{d_k} = \{1,2,3\}$$

$$\Big\langle p(s(a),Y)^{(1)}_{id} \,|\, p(s(a),Y)^{(2)}_{id} \,[\![\, p(W_1,W_2)^{(1)}_{id,[1],p(W_1,W_2),\gamma,[W_1]} \,|\, p(W_1,W_2)^{(2)}_{id,[2],p(W_1,W_2),\gamma,[W_1]} \Big\rangle$$

$$\text{unfold} \quad \Big|$$

$$\langle \mathsf{true}_{\{Y \to a\}} \,|\, p(s(a),Y)_{id} \,[\![\, \mathsf{true}_{\sigma,[1],p(W_1,W_2),\gamma \wedge \hat{\sigma},[W_1]} \,|\, p(W_1,W_2)_{id,[2],p(W_1,W_2),\gamma,[W_1]} \rangle$$

$$\text{success} \quad \Big|$$

$$\langle \textsc{success}_{id} \,[\![\, \textsc{success}_{id} \rangle$$

$$\gamma = \forall\, W_2, X, Y \quad p(W_1,W_2) \neq p(f(X),s(Y))$$
$$\sigma = \{W_1 \to s(W'),$$
$$\quad W_2 \to a\}$$
$$\hat{\sigma} = (W_1 = s(W')) \wedge (W_2 = a)$$
$$Traces \leftarrow Traces \cup \{[1]\}$$
$$TestCases \leftarrow TestCases \cup \{\}$$

Figure 4.6: Execution path of $p(s(a), Y)$

The execution in Figure 4.4 leads to the creation of a new goal: $p(s(c), Y)$. The corresponding execution is depicted by Figure 4.7.

$$\Big\langle p(s(c),Y)_{id} \,[\![\, p(W_1,W_2)_{id,[],p(W_1,W_2),\mathsf{true},[W_1]} \Big\rangle$$

$$\text{choice} \;\Big|\; \overline{c_n} = \{2\}, \overline{d_k} = \{1,2,3\}$$

$$\Big\langle p(s(c),Y)^{(2)}_{id} \,[\![\, p(W_1,W_2)^{(2)}_{id,[2],p(W_1,W_2),\gamma,[W_1]} \Big\rangle$$

$$\text{unfold} \;\Big|$$

$$\big\langle q(c)_{\{Y\to a\}} \,[\![\, q(W')_{\sigma,[2],p(W_1,W_2),\gamma\wedge\hat{\sigma},[W_1,W']} \big\rangle$$

$$\text{choice\_fail} \;\Big|\; \overline{c_n} = \{\}, \overline{d_k} = \{5,6\}$$

$$\big\langle \mathsf{fail}_{\{Y\to a\}} \,[\![\, \mathsf{fail}_{\sigma,[2],p(W_1,W_2),\gamma\wedge\hat{\sigma}\wedge\gamma',[W_1,W']} \big\rangle$$

$$\text{failure} \;\Big|$$

$$\langle \textsc{fail}_{id} \,[\![\, \textsc{fail}_{id} \rangle$$

$$\begin{aligned}
\gamma &= \forall\, W_2, X, Y \quad p(W_1,W_2) \neq p(s(a),b) \\
&\qquad\qquad \wedge \quad p(W_1,W_2) \neq p(f(X),s(Y)) \\
\sigma &= \{W_1 \to s(W'), \\
&\qquad W_2 \to a\} \\
\hat{\sigma} &= (W_1 = s(W')) \wedge (W_2 = a) \\
\gamma' &= \quad q(W') \neq q(a) \\
&\qquad \wedge \quad q(W') \neq q(b) \\
\mathit{Traces} &\leftarrow \mathit{Traces} \\
\mathit{TestCases} &\leftarrow \mathit{TestCases} \cup \{\}
\end{aligned}$$

Figure 4.7: Execution path of $p(s(c), Y)$

Figure 4.8 and 4.9 show the concolic execution of $p(f(b), Y)$ and $p(f(c), Y)$, which had been discovered in the execution of Figure 4.5.

$$\langle p(f(b),Y)_{id} \rangle\!\rangle\ p(W_1,W_2)_{id,[],p(W_1,W_2),\mathsf{true},[W_1]} \rangle$$

$$\text{choice} \quad \Big| \quad \overline{c_n} = \{3\}, \overline{d_k} = \{1,2,3\}$$

$$\left\langle p(f(b),Y)^{(3)}_{id} \right\rangle\!\rangle\ p(W_1,W_2)^{(3)}_{id,[3],p(W_1,W_2),\gamma,[W_1]} \Big\rangle$$

$$\text{unfold} \quad \Big|$$

$$\langle r(b,Y')_{\{Y \to s(Y')\}} \rangle\!\rangle\ r(W_1',W_2')_{\sigma,[3],p(W_1,W_2),\gamma \wedge \hat{\sigma},[W_1,W_1',W_2']} \rangle$$

$$\text{choice\_fail} \quad \Big| \quad \overline{c_n} = \{\}, \overline{d_k} = \{8,9\}$$

$$\langle \mathsf{fail}_{\{Y \to s(Y')\}} \rangle\!\rangle\ \mathsf{fail}_{\sigma,[3],p(W_1,W_2),\gamma \wedge \hat{\sigma} \wedge \gamma',[W_1,W_1',W_2']} \rangle$$

$$\text{failure} \quad \Big|$$

$$\langle \text{FAIL}_{id} \rangle\!\rangle\ \text{FAIL}_{id} \rangle$$

$$\begin{aligned}
\gamma &= \forall\ W_2, X \quad p(W_1,W_2) \neq p(s(a),b) \\
&\qquad\ \wedge \quad p(W_1,W_2) \neq p(s(X),a) \\
\sigma &= \{W_1 \to f(W_1'), \\
&\qquad W_2 \to s(W_2')\} \\
\hat{\sigma} &= (W_1 = f(W_1')) \wedge (W_2 = s(W_2')) \\
\gamma' &= \forall\ W_2' \quad r(W_1',W_2') \neq r(a,b) \\
&\qquad\ \wedge \quad r(W_1',W_2') \neq r(c,b) \\
&\mathit{Traces} \leftarrow \mathit{Traces} \\
&\mathit{TestCases} \leftarrow \mathit{TestCases} \cup \{\}
\end{aligned}$$

Figure 4.8: Execution path of $p(f(b), Y)$

$$\Big\langle p(f(c),Y)_{id} \;[\![\; p(W_1,W_2)_{id,[],p(W_1,W_2),\mathsf{true},[W_1]} \Big\rangle$$

$$\text{choice} \;\Big|\; \overline{c_n} = \{3\}, \overline{d_k} = \{1,2,3\}$$

$$\Big\langle p(f(c),Y)^{(3)}_{id} \;[\![\; p(W_1,W_2)^{(3)}_{id,[3],p(W_1,W_2),\gamma,[W_1]} \Big\rangle$$

$$\text{unfold} \;\Big|$$

$$\Big\langle r(c,Y')_{\{Y \to s(Y')\}} \;[\![\; r(W_1',W_2')_{\sigma,[3],p(W_1,W_2),\gamma\wedge\hat{\sigma},[W_1,W_1',W_2']} \Big\rangle$$

$$\text{choice} \;\Big|\; \overline{c_n} = \{9\}, \overline{d_k} = \{8,9\}$$

$$\Big\langle r(c,Y')^{(9)}_{\{Y \to s(Y')\}} \;[\![\; r(W_1',W_2')^{(9)}_{\sigma,[3,9],p(W_1,W_2),\gamma\wedge\hat{\sigma}\wedge\gamma',[W_1,W_1',W_2']} \Big\rangle$$

$$\text{unfold} \;\Big|$$

$$\Big\langle \mathsf{true}_{\left\{\begin{smallmatrix} Y \to s(b) \\ Y' \to b \end{smallmatrix}\right\}} \;[\![\; \mathsf{true}_{\sigma',[3,9],p(W_1,W_2),\gamma\wedge\hat{\sigma}\wedge\gamma'\wedge\hat{\sigma}',[W_1,W_1',W_2']} \Big\rangle$$

$$\text{success} \;\Big|$$

$$\Big\langle \textsc{success}_{\left\{\begin{smallmatrix} Y \to s(b) \\ Y' \to b \end{smallmatrix}\right\}} \;[\![\; \textsc{success}_{id} \Big\rangle$$

$$\begin{aligned}
&\gamma = \forall\ W_2, X \quad p(W_1,W_2) \neq p(s(a),b) \\
&\qquad\qquad \wedge \quad p(W_1,W_2) \neq p(s(X),a) \\
&\sigma = \{W_1 \to f(W_1'), \\
&\qquad W_2 \to s(W_2')\} \\
&\hat{\sigma} = (W_1 = f(W_1')) \wedge (W_2 = s(W_2')) \\
&\gamma' = \forall\ W_2' \quad r(W_1',W_2') \neq r(a,b) \\
&\sigma' = \{W_1' \to c, \\
&\qquad W_2' \to b\} \\
&\hat{\sigma}' = (W_1' = c) \wedge (W_2' = b) \\
&\mathit{Traces} \leftarrow \mathit{Traces} \cup \{[3,9]\} \\
&\mathit{TestCases} \leftarrow \mathit{TestCases} \cup \{\}
\end{aligned}$$

Figure 4.9: Execution path of $p(f(c), Y)$

Since we have no more test case to perform, the execution stops. In this particular case, the backtracking rule is never needed. This comes from the fact that this rule is only performed when two conditions are satisfied:

1. One concrete goal matches with several clauses.

2. The first matching clauses produces a failing path.

In this example, the first condition is only verified once and the second is not, so the rule is not applicable.

The final list of produced test cases is thus:

$$\begin{aligned} \mathit{TestCases} = \{&p(a, Y),\ p(s(b), W_2),\ p(f(a), W_2),\ p(s(a), W_2), p(s(c), W_2),\\ &p(f(b), W_2),\ p(f(c), W_2)\} \end{aligned}$$

Note that the second argument of p is still a variable. It is in fact the desired behaviour, since we asked only for the first variable to be grounded.

Chapter 5

Ongoing Implementation

This chapter presents an overview of our current implementation. First we describe the execution environment, the used tools and other home-made files. Then, we focus on the algorithm implementation itself.

5.1 The Working Environment

All the implementation is based on SWI-Prolog (SWI-Prolog 2019; Wielemaker et al. 2012) which provides a few interesting features, like a C interface and methods to deal with external input files.

The program is composed of five files and uses several existing libraries as shown in Figure 5.1.

Figure 5.1 displays a class diagram showing all the files we used. The central file of this implementation is *concolic_tool.pl*, which contain all the concolic testing procedure. It uses three interfaces: *swiplz3.pl*, *z3_parser.pl* and *prolog_reader.pl*. In each home-made class (i.e. all of them except the C libraries), one can find the list of all public methods, available for the other files.

z3_parser.pl only proposes one public method to transform a String containing a term in Z3-form to a classic Prolog term. *prolog_reader.pl* provides some functions to deal with user input and files.

swiplz3.c calls several standard C libraries. We are only interested in two of them here: *z3.h* which is the Z3 API for the C language, and *SWI-Prolog.h* which is the library to combine C and Prolog. *swiplz3.c* works with *swiplz3.pl*. This file creates the missing functions relative to Z3 and make the C functions available for the other Prolog files. For clarity, we have not copied the C functions that the Prolog interface made available for the rest of the program.

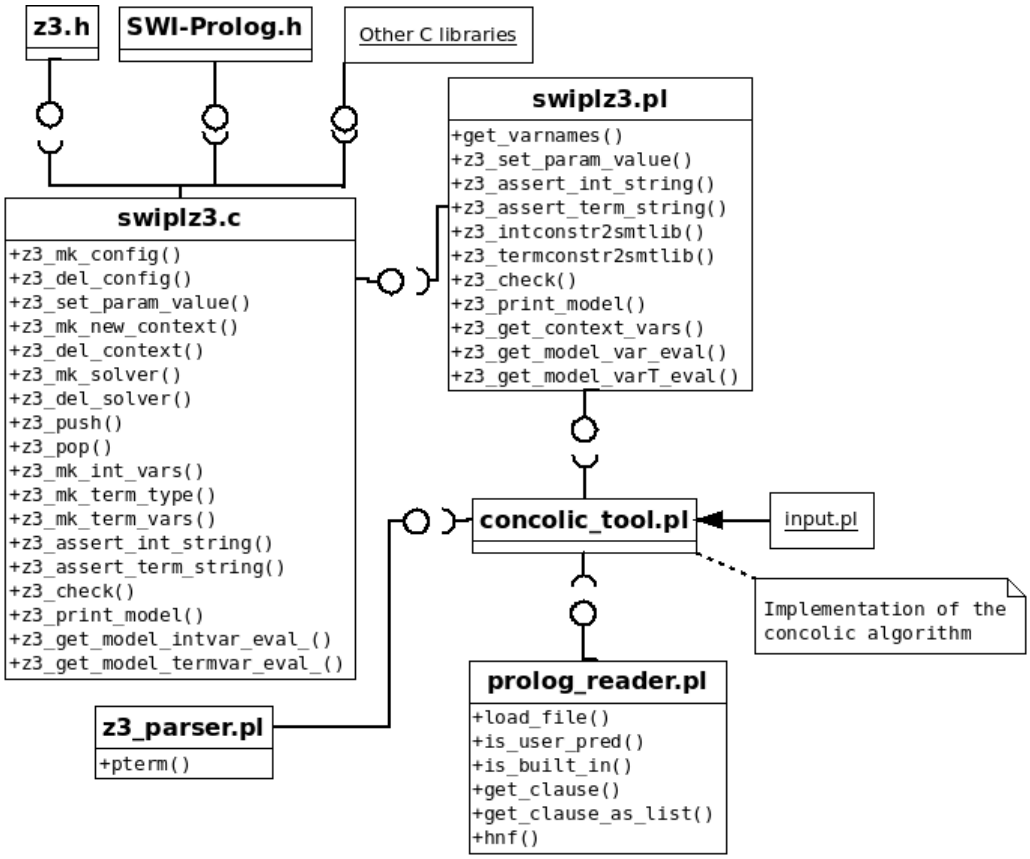

Figure 5.1: Class diagram of the implementation

Both *swiplz3.c* and *swiplz3.pl* aim to offer Z3 functionalities to *concolic_tool.pl*. This means that they need to parse Prolog types and constraints into Z3 types and constraints, and vice versa. Currently, they only work with two types of data: Prolog terms and regular integers.

The file *input.pl* represents the input Prolog program we want to test. We will see in the last section of this chapter the two ways to call it.

5.2 The Core Implementation

In this section, we describe the principal file, called *concolic_tool.pl*.

As in the version proposed in the previous chapter, the one in Figure 5.2 needs an initial (concrete) atomic goal, $p(s(a), X)$ for example. The initial configuration is then:

$$\langle p(s(a), X)_{id} \, [\![\, p(W_1, W_2)_{id,\{\,\},p(W_1,W_2),true,W_1} \rangle$$

$$\text{(success)}\ \frac{}{\langle \mathsf{true}_{\delta} \,[\![\, \mathsf{true}_{\theta,\pi,A_0,\gamma,G} \rangle \rightsquigarrow \langle \textsc{success}_{\delta} \,[\![\, \textsc{success}_{\theta} \rangle}$$

$$\text{(unfold)}\ \frac{\begin{array}{l}\mathsf{clauses}(A,\mathcal{P}) = \overline{c_n} \wedge n > 0 \wedge \mathsf{clauses}(A',\mathcal{P}) = \overline{d_k} \\ c_i = (H_i \leftarrow \mathcal{B}_i), i \in \{1,\ldots,n\}, \mathsf{mgu}(A,H_1) = \sigma \wedge \mathsf{mgu}(A',H_1) = \rho \\ \gamma' \leftarrow \mathit{neg\_constr}(A', \overline{d_k} \setminus \overline{c_n}) \\ \mathit{if}\ \pi \notin \mathit{Traces}\ \mathit{then}\ \mathit{TestCases} \leftarrow \mathit{TestCases} \cup \mathsf{alts}(A_0,\gamma,A',\overline{c_n},\overline{d_k},G) \\ \mathit{Traces} \leftarrow \mathit{Traces} \cup \pi,\end{array}}{\begin{array}{l}\langle (A,\mathcal{B})_{\delta} \,[\![\, (A',\mathcal{B}')_{\theta,\pi,A_0,\gamma,G} \rangle \rightsquigarrow \langle (\mathcal{B}_1\sigma, \mathcal{B}\sigma)_{\delta\sigma} \\ \qquad\qquad [\![\, (\mathcal{B}_1\rho, \mathcal{B}'\rho)_{\theta\rho,\pi.\ell(c_i),A_0,\gamma\wedge\gamma'\wedge\widehat{\rho},G\wedge\mathcal{V}ar(G\rho)} \rangle\end{array}}$$

$$\text{(failure)}\ \frac{\begin{array}{l}\mathsf{clauses}(A,\mathcal{P}) = \{\} \wedge \mathsf{clauses}(A',\mathcal{P}) = \overline{c_k} \\ \gamma' \leftarrow \mathit{neg\_constr}(A', \overline{c_k}) \\ \mathit{if}\ \pi \notin \mathit{Traces}\ \mathit{then}\ \mathit{TestCases} \leftarrow \mathit{TestCases} \cup \mathsf{alts}(A_0,\gamma,A',\{\ \},\overline{c_k},G) \\ \mathit{Traces} \leftarrow \mathit{Traces} \cup \pi,\end{array}}{\langle (A,\mathcal{B})_{\delta} \,[\![\, (A',\mathcal{B}')_{\theta,\pi,A_0,\gamma,G} \rangle \rightsquigarrow \langle (\mathsf{fail},\mathcal{B})_{\delta} \,[\![\, (\mathsf{fail},\mathcal{B}')_{\theta,\pi,A_0,\gamma\wedge\gamma',G} \rangle}$$

Figure 5.2: Concolic execution semantics: implicit nondeterminism

where the first argument needs to be grounded. The two global parameters are set as usual:

$$\begin{aligned} \mathit{Traces} &\leftarrow \{\ \} \\ \mathit{TestCases} &\leftarrow \{p(s(a), X)\} \end{aligned}$$

The algorithm which is implemented in practice (Figure 5.2) has suffered a few changes from the one presented in the previous chapter. The differences come from the fact that we take advantage of internal Prolog mechanisms like backtracking. In the other version, nondeterminism is explicit. In this version, it becomes implicit. The consequence of this modification is that we can now combine choice with unfold into one single rule, and rules choice_fail with failure into another one. Using internal backtracking also means that we don't need to consider sequences of goals. Moreover, the selection of the clause in the new unfold rule is done non-terministically. This rule becomes thus non-deterministic, as it is usual in the definition of SLD resolution.

5.3 How to Use It?

Currently, the implementation is still in the development phase and so far, the results are promising. It should be soon available publicly. Once online, it will be possible for anybody to try it with some provided small examples, or to test his or her own Prolog program. This section aims to explain how to configure and use our implementation.

Before using the test implementation, the SWIPrologZ3 interface must be installed. This requires a few manipulations because the C source file must be compiled independently. Once it is done, the Z3 functions can be used in Prolog files.

The concolic testing tool can be used in two ways:

- The interactive mode: The simplest version, where we just load the program into SWI-Prolog and then call a predefined example. A folder with twenty Prolog examples is provided with the program. At the end of *concolic_tool.pl* one can find the predefined calls using those examples.

- The command mode: This more flexible version allows the user to save the program as a stand alone executable. He can then call it like any other executable and personalize the call with the parameters of his own choice. For example the call:

```
./concolic_tool -cg "p(s(a))" -ground "[1]"
-depth "2" -timeout "10" -file "examples/ex01.pl"
```

  means that:

  - $p(s(a))$ is the concrete initial goal
  - Only the first variable must be grounded
  - The depth of a path execution is maximum 2
  - The program stops after 10 seconds
  - The file *examples/ex01.pl* is the input program

Some other features are also available as for example the possibilities of giving the symbolic goal as well, or an option to add the computed trace for each test case produced.

Chapter 6

Conclusion and Future Developments

In this master thesis, we presented a new algorithm to test Prolog programs. This algorithm uses the method called "concolic testing" to combine both symbolic and concrete execution. It aims to generate test cases in a fully automatic way and offering good code coverage. We used a new coverage criteria, specific for logic programming, called "choice coverage". By using Z3 solver and SAT constraints, we have bypassed the unification problem. We have also combined some steps of the procedure in order to avoid some redundant computations. The complete step-by-step execution gives us a good hint about the soundness of our method. A proof-of-concept implementation will be soon publicly available to show the usefulness of the approach.

As future developments, we could remove some of the restrictions we made until now. We could develop the Z3 interface to deal with other types like float for example, and to accept more Z3-functions. In the current implementation, only the function we needed to use directly were implemented. In another context, someone could be interested to have more capabilities, already provided in the C API but not in our Prolog interface.

Another restriction, still in the implementation but also concerning the theory of our method is the definition of Prolog. The algorithm could be extend to full Prolog, to deal with cut and negation. Actually, this extension has been discussed in the first online appendix of Mesnard et al. (2015). The search should be continued after being adapted to our algorithm.

Bibliography

Albert, E., P. Arenas, M. Gómez-Zamalloa, and J.M. Rojas (2014). "Test Case Generation by Symbolic Execution: Basic Concepts, a CLP-Based Instance, and Actor-Based Concurrency". In: *SFM 2014*. Springer LNCS 8483, pp. 263–309.

Backus, John W et al. (1957). "The FORTRAN automatic coding system". In: *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*. ACM, pp. 188–198.

Belli, Fevzi and Oliver Jack (1993). "Implementation-Based Analysis and Testing of Prolog Programs". In: *ISSTA*, pp. 70–80.

Carlsson, Mats and Per Mildner (2012). "SICStus Prolog - The first 25 years". In: *Theory and Practice of Logic Programming* 12.1-2, pp. 35–66. DOI: 10.1017/S1471068411000482. URL: http://dx.doi.org/10.1017/S1471068411000482.

Cherep Dragoevich, Manuel (2016). "A Methodology for Applying Concolic Testing". MA thesis. Uppsala University.

Claessen, K. and J. Hughes (2000). "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Proc. of (ICFP 2000)*. ACM, pp. 268–279.

Clarke, L.A. (1976). "A program testing system". In: *Proceedings of the 1976 Annual Conference (ACM'76)*, pp. 488–491.

Clocksin, William F and Christopher S Mellish (2012). *Programming in Prolog: Using the ISO standard*. Springer Science & Business Media.

Colmerauer, Alain and Philippe Roussel (1996). "The birth of Prolog". In: *History of programming languages—II*. ACM, pp. 331–367.

Degrave, François, Tom Schrijvers, and Wim Vanhoof (2008). "Automatic Generation of Test Inputs for Mercury". In: *Logic-Based Program Synthesis and Transformation, 18th International Symposium, LOPSTR 2008*, pp. 71–86.

Giantsios, Aggelos, Nikolaos Papaspyrou, and Konstantinos Sagonas (2017). "Concolic testing for functional languages". In: *Science of Computer Programming* 147, pp. 109–134.

Github (2019a). *Z3-C API*. URL: http://z3prover.github.io/api/html/group__capi.html (visited on 05/08/2019).
— (2019b). *Z3Prover*. URL: https://github.com/Z3Prover (visited on 05/08/2019).
Godefroid, P., N. Klarlund, and K. Sen (2005). "DART: directed automated random testing". In: *Proc. of PLDI'05*. ACM, pp. 213–223.
Godefroid, P., M.Y. Levin, and D.A. Molnar (2012). "SAGE: whitebox fuzzing for security testing". In: *CACM* 55.3, pp. 40–44.
Gómez-Zamalloa, M., E. Albert, and G. Puebla (2010). "Test case generation for object-oriented imperative languages in CLP". In: *TPLP* 10.4-6, pp. 659–674.
Hermenegildo, Manuel V., Francisco Bueno, Manuel Carro, Pedro López-García, Edison Mera, José F. Morales, and Germán Puebla (2012). "An overview of Ciao and its design philosophy". In: *TPLP* 12.1-2, pp. 219–252.
Jones, Simon Peyton (2003). *Haskell 98 language and libraries: the revised report*. Cambridge University Press.
King, James C. (1976). "Symbolic Execution and Program Testing". In: *CACM* 19.7, pp. 385–394.
Kowalski, Robert A (1988). "The early years of logic programming". In: *Communications of the ACM* 31.1, pp. 38–43.
Le Traon, Yves (2018). *Software Testing and Validation: introduction*. Université du Luxembourg.
Learn Testing (2019). *Practical Guide to Software Unit Testing*. URL: https://learntesting123.blogspot.com/2015/03/21-practical-guide-to-software-unit.html (visited on 05/12/2019).
Lloyd, J.W. (1987). *Foundations of Logic Programming*. 2nd Ed. Springer-Verlag, Berlin.
McCarthy, John (1959). "Recursive functions of symbolic expressions and their computation by machine". In:
Mera, Edison, Pedro López-García, and Manuel V. Hermenegildo (2009). "Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework". In: *25th International Conference on Logic Programming, ICLP 2009, Pasadena*, pp. 281–295.
Mesnard, Frédéric, Étienne Payet, and Germán Vidal (2015). "Concolic testing in logic programming". In: *Theory and Practice of Logic Programming* 15.4-5, pp. 711–725.
— (2016). "On the Completeness of Selective Unification in Concolic Testing of Logic Programs". In: *LOPSTR*.
— (2017). "Selective unification in constraint logic programming". In: *PPDP*.

Microsoft Research (2019). *Z3*. URL: https://rise4fun.com/z3 (visited on 05/08/2019).

Palacios, Adrián and Germán Vidal (2015). "Concolic execution in functional programming by program instrumentation". In: *International Symposium on Logic-Based Program Synthesis and Transformation*. Springer, pp. 277–292.

Pasareanu, C.S. and N. Rungta (2010). "Symbolic PathFinder: symbolic execution of Java bytecode". In: *ASE*, pp. 179–180.

Schimpf, Joachim and Kish Shen (2012). "ECL^{i}PS^e - From LP to CLP". In: *Theory and Practice of Logic Programming* 12.1-2, pp. 127–156. DOI: 10.1017/S1471068411000469. URL: http://dx.doi.org/10.1017/S1471068411000469.

Sen, K., D. Marinov, and G. Agha (2005). "CUTE: a concolic unit testing engine for C". In: *Proc. of ESEC/SIGSOFT FSE 2005*. ACM, pp. 263–272.

Somogyi, Z., F. Henderson, and T. Conway (1996). "The execution algorithm of Mercury, an efficient purely declarative Logic Programming language". In: *The Journal of Logic Programming* 29.1–3, pp. 17–64.

Ströder, T., F. Emmes, P. Schneider-Kamp, J. Giesl, and C. Fuhs (2011). "A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog". In: *LOPSTR'11*. Springer LNCS 7225, pp. 237–252.

SWI-Prolog (2019). *SWI-Prolog*. URL: http://www.swi-prolog.org/ (visited on 05/22/2019).

Tikovsky, Jan Rasmus (2017). "Concolic Testing of Functional Logic Programs". In: *Declarative Programming and Knowledge Management*. Springer, pp. 169–186.

Vanhoof, Wim (2013). *De la programmation fonctionnelle vers la programmation logique*. Université de Namur.

Vasak, T. and J. Potter (1986). "Characterization of terminating logic programs". In: *Proc. of the 1986 Intl. Symp. on Logic Programming*. IEEE, pp. 140–147.

Vidal, G. (2015). "Concolic Execution and Test Case Generation in Prolog". In: *Proc. of the 24th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'14)*. Ed. by M. Proietti and H. Seki. Springer LNCS 8981, pp. 167–181.

Wielemaker, Jan, Tom Schrijvers, Markus Triska, and Torbjörn Lager (2012). "SWI-Prolog". In: *Theory and Practice of Logic Programming* 12.1-2, pp. 67–96. ISSN: 1471-0684.

Wikipedia (2019a). *Concolic testing*. URL: https://en.wikipedia.org/wiki/Concolic_testing (visited on 04/23/2019).

Wikipedia (2019b). *Symbolic execution*. URL: https://en.wikipedia.org/wiki/Symbolic_execution (visited on 05/10/2019).